Capitolo 6

Gestione delle eccezioni

di *David R. Chung*

IN QUESTO CAPITOLO

- Che cos'è un'eccezione? 133
- Se le eccezioni sono la risposta, qual è la domanda? 134
- Un po' di terminologia 136
- Generazione di un'eccezione 136
- Blocchi throw, try e catch 137
- La classe Throwable 141
- Tipi di eccezioni 142
- Eccezioni integrate 150
- Riepilogo 151

Gli errori sono parte normale della programmazione. Alcuni di essi riguardano la progettazione di base o l'implementazione di un programma e vengono spesso chiamati *bug*, altri invece non sono dei veri bug, ma il risultato di situazioni particolari, come memoria insufficiente o nomi di file non validi.

In base a come vengono gestiti, gli errori di questo secondo tipo possono diventare dei bug. Sfortunatamente, se l'obiettivo che ci si propone è creare applicazioni robuste, si passa più tempo a gestire gli errori che non a scrivere il cuore di un'applicazione.

Il meccanismo di gestione delle eccezioni di Java permette di gestire gli errori senza obbligare a spendere la maggior parte delle energie preoccupandosi di essi.

Che cos'è un'eccezione?

Come indica il nome, un'*eccezione* è una condizione eccezionale, qualcosa al di fuori dell'ordinario. Nella maggior parte dei casi le eccezioni vengono utilizzate per riportare condizioni di errore, ma possono anche essere uno strumento per indicare altre situazioni. Questo capitolo si concentra principalmente sulle eccezioni come meccanismo di gestione degli errori.

Le eccezioni costituiscono uno strumento per informare che vi sono degli errori e un modo per gestirli. Questa struttura di controllo permette di specificare esattamente dove gestire tipi specifici di errori.

Altri linguaggi, come C++ e Ada, includono la gestione delle eccezioni. Quella di Java è simile a quella utilizzata in C++.

Tennyson aveva capito il problema

Nel suo poema, *Charge of the light brigade*, Andrew Lord Tennyson descrive una battaglia in cui a una brigata di cavalleria viene ordinato di attaccare una postazione di cannoni. Si scopre che la vallata in cui avviene l'attacco è una trappola: su tre lati vi sono grossi cannoni e i coraggiosi soldati a cavallo con le sciabole vengono massacrati. Il poema descrive una vera battaglia avvenuta nella Guerra di Crimea.

La battaglia, nel modo in cui Tennyson la descrive, porta alla luce un problema classico. Qualcuno, probabilmente lontano dal fronte, aveva dato l'ordine di attaccare. Gli uomini che guidavano la carica si resero ben presto conto che era stato commesso un errore, ma sfortunatamente non avevano l'autorità di dire nulla a proposito. Nelle parole immortali di Tennyson, "Non potevano contestare, potevano solo eseguire e morire: i 600 cavalcarono nella Valle della Morte."

L'utilizzo delle eccezioni in Java permette di determinare esattamente chi gestisce un errore. Infatti, le funzioni a basso livello possono rilevare gli errori, mentre le funzioni di livello più alto decidono che cosa farne. Le eccezioni costituiscono uno strumento per comunicare informazioni lungo la catena di metodi, finché uno di questi può gestirli.

Se le eccezioni sono la risposta, qual è la domanda?

La maggior parte dei linguaggi procedurali come C e Pascal non utilizza la gestione degli errori. In questi linguaggi vengono utilizzate diverse tecniche per determinare se è avvenuto un errore; il metodo più comune consiste nel controllare il valore restituito da una funzione.

Si supponga di dover calcolare e visualizzare il prezzo di vendita al dettaglio di un articolo. In questo esempio, il prezzo al dettaglio è il doppio del prezzo di vendita all'ingrosso:

```
int costoDettaglio( int costoIngrosso ) {
    if (costoIngrosso <= 0 ) {
        return 0 ;
    }
    return (costoIngrosso * 2 ) ;
}
```

Il metodo `costoDettaglio()` utilizza il prezzo all'ingrosso di un articolo e lo raddoppia. Se il prezzo all'ingrosso è un numero negativo o nullo, la funzione restituisce zero per indicare

che è avvenuto un errore. Questo metodo può essere utilizzato in un'applicazione nel seguente modo:

```
int costoIngrosso  = 30 ;
int costoDettaglio = 0 ;
costoDettaglio = costoDettaglio( costoIngrosso ) ;
System.out.println( "Prezzo all'ingrosso = " + costoIngrosso ) ;
System.out.println( "Prezzo al dettaglio = " + costoDettaglio ) ;
```

In questo esempio, il metodo `costoDettaglio()` calcola il prezzo al dettaglio corretto e lo stampa. Il problema è che il codice non controlla mai se la variabile `costoIngrosso` è negativa. Anche se il metodo controlla il valore di `costoIngrosso` e riporta un errore, non vi è nulla che obblighi il metodo che ha effettuato la chiamata a gestire l'errore. Se questo metodo viene richiamato con un valore di `costoIngrosso` negativo, la funzione stampa alla cieca i dati errati. Di conseguenza, non ha importanza quanto diligenti si è nell'assicurarsi che il metodo restituisca valori che indicano un errore: i metodi che hanno effettuato la chiamata sono liberi di ignorarli.

È possibile evitare che vengano stampati valori errati inserendo l'intera operazione in un metodo. Il metodo `mostraDettaglio()` utilizza il prezzo all'ingrosso, lo raddoppia e lo stampa; se è negativo o zero, il metodo non stampa niente e restituisce il valore booleano `false`:

```
boolean mostraDettaglio( int costoIngrosso ) {
    if ( costoIngrosso <= 0 ) {
        return false ;
    }
    int costoDettaglio ;
    costoDettaglio = costoIngrosso * 2 ;
    System.out.println( "Prezzo all'ingrosso = " + costoIngrosso ) ;
    System.out.println( "Prezzo al dettaglio = " + costoDettaglio ) ;
    return true ;
}
```

Utilizzando questo nuovo metodo si garantisce che non vengano mai stampati valori errati. Tuttavia, ancora una volta, il metodo che ha effettuato la chiamata non deve controllare se viene restituito `true`.

Il fatto che chi ha effettuato la chiamata possa scegliere di ignorare i valori restituiti non è l'unico problema che si riscontra utilizzando i valori restituiti per il rilevamento degli errori. Che cosa succede se un metodo restituisce un valore booleano e sia `true` che `false` sono valori validi? In che modo questo metodo riporta un errore? Si consideri un metodo per determinare se uno studente ha superato un esame. Il metodo `pass()` utilizza il numero di risposte corrette e il numero di domande. Il metodo calcola la percentuale: se è superiore al 70%, lo studente è promosso. Si consideri il metodo `votoPromosso()`:

```
boolean votoPromosso( int corretto, int totale ) {
    boolean codiceRest = false ;
    if ( (float)corretto / (float)totale > 0.70 ) {
        codiceRest = true ;
    }
    return codiceRest ;
}
```

Questo esempio funziona finché gli argomenti del metodo sono corretti. Che cosa succede se il numero di risposte corrette è maggiore del totale o, peggio ancora, se il totale è zero (cosa che causa una divisione per zero nel metodo)? Basandosi sui valori restituiti, in questo caso non vi è modo di riportare un errore in questa funzione.

Le eccezioni permettono di evitare che i valori restituiti abbiano questa doppia funzione e permettono di utilizzarli solo come informazioni utili dai metodi. Le eccezioni inoltre obbligano il metodo che ha effettuato la chiamata a gestire gli errori, in quanto non possono essere ignorate.

Un po' di terminologia

La gestione degli errori può essere considerata come una struttura di controllo non locale. Quando un metodo *genera* un'eccezione, il metodo che lo ha richiamato deve determinare se è in grado di *intercettare* l'eccezione. In caso positivo, il metodo che ha effettuato la chiamata assume il controllo e l'esecuzione continua da quel punto; diversamente, l'eccezione viene passata al metodo che ha effettuato la chiamata. Questo processo continua finché l'eccezione viene intercettata o finché viene raggiunta la cima dello stack (o il fondo, a seconda di come lo si guarda) e l'applicazione termina.

Le eccezioni di Java sono oggetti di classe derivati da `java.lang.Throwable`, e come tali possono contenere sia dati che metodi. Infatti, la classe di base `Throwable` implementa un metodo che restituisce una `String` che descrive l'errore che ha causato l'eccezione. Ciò risulta utile per il debugging e se si desidera fornire agli utenti un messaggio di errore esplicativo.

Generazione di un'eccezione

Il metodo `votoPromosso()` presentato nel paragrafo precedente non era in grado di riportare una condizione di errore, in quanto tutti i possibili valori restituiti erano validi. Aggiungendo la gestione delle eccezioni al metodo è possibile separare il riporto dei risultati dal riporto degli errori.

Il primo passaggio consiste nel modificare la definizione del metodo `votoPromosso()` in modo da includere la clausola `throws`, che fornisce un elenco dei tipi di eccezioni che possono essere generati dal metodo. Nel seguente codice modificato, il metodo genera solo un'espressione del tipo `Exception`:

```
static boolean voroPromosso( int corretto, int totale ) throws Exception {
    boolean codiceRest = false ;
```

Il resto del metodo rimane per lo più invariato. Questa volta, il metodo controlla se gli argomenti hanno senso. Poiché occorre determinare la percentuale di risposte corrette, sarebbe irragionevole avere più risposte corrette che risposte totali e pertanto, in presenza di una simile situazione, il metodo genera un'eccezione. Il metodo crea l'istanza di un oggetto di tipo `Exception`. Il costruttore `Exception` utilizza un parametro `String`, che contiene un messaggio che può essere richiamato quando viene intercettata l'eccezione.

L'istruzione throw termina il metodo e consente di intercettarlo:

```
        if( corretto > totale ) {
            throw new Exception( "Valori non validi" ) ;
        }
        if ( (float)corretto / (float)totale > 0.70 ) {
            codiceRest = true ;
        }
        return codiceRest ;
    }
```

Blocchi throw, try e catch

Per rispondere a un'eccezione, la chiamata al metodo che l'ha prodotta deve trovarsi all'interno di un blocco try, che è un blocco di codice che inizia con la parola chiave try seguita da una parentesi graffa aperta e da una chiusa. Tutti i blocchi try sono associati a uno o più blocchi catch e si presentano come indicato di seguito:

```
    try
        {
        // qui vanno le chiamate ai metodi
        }
```

Se un metodo deve intercettare le eccezioni generate dai metodi che richiama, le chiamate devono essere inserite all'interno di un blocco try. Se viene generata un'eccezione, questa viene gestita in un blocco catch. Diversi blocchi catch gestiscono diversi tipi di eccezioni. Di seguito sono riportati un blocco try e un blocco catch impostati in modo da gestire le eccezioni di tipo Exception:

```
    try
        {
        // qui vanno le chiamate ai metodi
        }
    catch( Exception e )
        {
        // qui va la gestione delle eccezioni
        }
```

Quando un metodo qualsiasi nel blocco try genera un qualsiasi tipo di eccezione, l'esecuzione del blocco try cessa e il controllo del programma passa immediatamente al blocco catch associato. Se questo è in grado di gestire quel tipo di eccezione, assume il controllo dell'esecuzione, altrimenti l'eccezione viene passata a chi ha richiamato il metodo. In un'applicazione, questo processo continua finché un blocco catch intercetta l'eccezione o finché questa raggiunge il metodo main() senza essere stata intercettata, facendo terminare l'applicazione.

Un esempio di eccezione

Poiché tutti i metodi di Java sono membri di classe, il metodo votoPromosso() viene incorporato nella classe votoEsame. Poiché main() richiama votoPromosso(), main() deve essere in grado di intercettare qualsiasi eccezione che votoPromosso() potrebbe generare.

Per fare ciò, `main()` inserisce la chiamata a `votoPromosso()` in un blocco `try`. Poiché la clausola `throws` include il tipo `Exception`, il blocco `catch` intercetta la classe `Exception`. Il Listato 6.1 mostra l'intera applicazione `votoEsame`.

Listato 6.1 *L'applicazione votoEsame.*

```
import java.io.* ;
import java.lang.Exception ;
public class votoEsame {
    public static void main( String[] args ) {
        try
            {
            // la seconda chiamata a votoPromosso genera
            // un'eccezione, perciò la terza chiamata
            // non viene mai eseguita
            System.out.println( votoPromosso( 60,  80 ) ) ;
            System.out.println( votoPromosso( 75,   0 ) ) ;
            System.out.println( votoPromosso( 90, 100 ) ) ;
            }
        catch( Exception e )
            {
            System.out.println( "Intercettata eccezione -" +
                                e.getMessage() ) ;
            }
    }
    static boolean votoPromosso( int corretto, int totale )
                                        throws Exception {
        boolean codiceRest = false ;
        if( corretto > totale ) {
            throw new Exception( "Valori non validi" ) ;
        }
        if ( (float)corretto / (float)totale > 0.70 ) {
            codiceRest = true ;
        }
        return codiceRest ;
    }
}
```

La seconda chiamata a `votoPromosso()` in questo caso fallisce perché il metodo controlla se il numero di risposte corrette è inferiore al numero totale delle risposte. Quando `votoPromosso()` genera un'eccezione, il controllo passa al metodo `main()`. In questo esempio, il blocco `catch` in `main()` intercetta l'eccezione e stampa "Intercettata eccezione – Valori non validi".

Blocchi catch multipli

In alcuni casi, può succedere che un metodo debba intercettare diversi tipi di eccezioni. In Java è possibile utilizzare diversi blocchi `catch`, ognuno dei quali deve specificare un tipo diverso di eccezione:

```
try
    {
    // qui vanno le chiamate ai metodi
    }
catch( UnaClasseException e )
    {
    // qui sono gestite le eccezioni UnaClasseException
    }
catch( UnAltraClasseException e )
    {
    // qui sono gestite le eccezioni UnAltraClasseException
}
```

Quando nel blocco `try` viene generata un'eccezione, questa viene intercettata dal primo blocco `catch` del tipo appropriato. Viene eseguito un solo blocco `catch` di una determinata serie. Si noti che i blocchi `catch` assomigliano alle dichiarazioni dei metodi. L'eccezione intercettata in un blocco `catch` è un riferimento locale al vero oggetto eccezione, utilizzabile per determinare che cosa ha generato inizialmente l'eccezione.

Tutti i metodi devono intercettare tutte le eccezioni?

Che cosa succede se un metodo richiama un altro metodo che genera un'eccezione, ma che sceglie di non intercettarla? Nell'esempio nel Listato 6.2, `main()` richiama `foo()`, che a sua volta richiama `bar()`. Questo include `Exception` nella clausola `throws`; poiché `foo()` non può intercettare l'eccezione, deve includere nella clausola `throws` anche `Exception`. L'applicazione del Listato 6.2 mostra un metodo, `foo()`, che ignora le eccezioni generate dal metodo richiamato.

Listato 6.2 *Un metodo che ignora le eccezioni generate dal metodo richiamato.*

```
import java.io.* ;
import java.lang.Exception ;
public class MultiThrow {
    public static void main( String[] args ) {
        try
            {
            foo() ;
            }
        catch( Exception e )
            {
            System.out.println( "Intercettata eccezione " +
                                e.getMessage() ) ;
            }
    }
    static void foo() throws Exception {
        bar() ;
    }
    static void bar() throws Exception {
        throw new Exception( "Chi se ne importa" ) ;
    }
}
```

Nell'esempio nel Listato 6.3, `main()` richiama `foo()` che richiama `bar()`. Poiché `bar()` genera un'eccezione e non la intercetta, `foo()` ha l'opportunità di intercettarla. Il metodo `foo()` non ha blocchi `catch`, pertanto non può intercettare l'eccezione. In questo caso, l'eccezione si propaga verso la cima dello stack al metodo `main()` che ha richiamato `foo()`.

Listato 6.3 *Un metodo che intercetta e rigenera un'eccezione.*

```
import java.io.* ;
import java.lang.Exception ;
public class MultiThrow {
    public static void main( String[] args ) {
        try
            {
            foo() ;
            }
        catch( Exception e )
            {
            System.out.println( "Intercettata eccezione " +
                                e.getMessage() ) ;
            }
    }
    static void foo() throws Exception {
        try
            {
            bar() ;
            }
        catch( Exception e )
            {
            System.out.println( "Rigenere l'eccezione - " +
                                e.getMessage() ) ;
            throw e ;
            }
    }
    static void bar() throws Exception {
        throw new Exception( "Chi se ne importa" ) ;
    }
}
```

Il metodo `foo()` richiama `bar()`, che genera un'eccezione che viene intercettata da `foo()`. In questo esempio, `foo()` semplicemente *rigenera* l'eccezione, che viene infine intercettata nel metodo `main()` dell'applicazione. In un'applicazione reale, `foo()` potrebbe eseguire del codice e quindi rigenerare l'eccezione. Questo permette sia a `foo()` che a `main()` di gestire l'eccezione.

La clausola finally

Java introduce un nuovo concetto nella gestione delle eccezioni: la clausola `finally`, che contraddistingue un blocco di codice che viene sempre eseguito. Ecco un esempio:

```
import java.io.* ;
import java.lang.Exception ;
```

```
public class MultiThrow {
    public static void main( String[] args ) {
        try
            {
            alpha() ;
            }
        catch( Exception e }
            {
            System.out.println( "Intercettata eccezione " ) ;
            }
        finally()
            {
            System.out.println( "Finalmente. " ) ;
            }
    }
}
```

Nell'esecuzione normale, vale a dire quando non vengono generate eccezioni, il blocco `finally` viene eseguito immediatamente dopo il blocco `try`. Quando viene generata un'eccezione, il blocco `finally` viene eseguito prima che il controllo passi al metodo che ha effettuato la chiamata.

Se `alpha()` genera un'eccezione, questa viene intercettata nel blocco `catch` e successivamente viene eseguito il blocco `finally`. Se `alpha()` non genera alcuna eccezione, il blocco `finally` viene eseguito dopo il blocco `try`. Anche se viene eseguita solo una parte del codice di un blocco `try`, il blocco `finally` viene comunque eseguito.

La classe Throwable

Tutte le eccezioni in Java sono sottoclassi della classe `Throwable`. Se si desidera creare una classe di eccezioni personalizzata, è necessario creare una sottoclasse di `Throwable`. Per la maggior parte dei programmi di Java non è necessario creare sottoclassi delle classi di eccezioni.

Di seguito è presentata la parte `public` della definizione di classe di `Throwable`:

```
public class Throwable {
    public Throwable() ;
    public Throwable(String message) ;
    public String getMessage()
    public String toString() ;
    public void printStackTrace() ;
    public void printStackTrace(
                        java.io.PrintStream s) ;
    private native void printStackTrace0(
                        java.io.PrintStream s);
    public native Throwable fillInStackTrace();
}
```

Il costruttore utilizza una stringa che descrive l'eccezione. Quando viene generata un'eccezione, è possibile richiamare il metodo `getMessage()` per ottenere la stringa dell'errore.

Tipi di eccezioni

Anche i metodi dell'API di Java e il linguaggio stesso generano eccezioni, che possono essere suddivise in due classi: `Exception` ed `Error`.

Entrambe queste classi derivano da `Throwable`. `Exception` e le relative sottoclassi vengono utilizzate per indicare condizioni che possono essere corrette, `Error` e le relative sottoclassi indicano condizioni che generalmente non possono essere corrette e che fanno terminare l'applet.

I diversi package inclusi nel Java Development Kit generano diversi tipi di eccezioni `Exception` ed `Error`, descritte nel seguito.

Eccezioni di java.awt

Le classi dell'AWT hanno membri che generano un errore e due eccezioni:

- ✔ `AWTException` (eccezione nell'AWT).
- ✔ `IllegalComponentStateException` (un componente non è nello stato idoneo per un'operazione richiesta).
- ✔ `AWTError` (errore nell'AWT).

Eccezione di java.awt.datatransfer

Le classi del package di trasferimento dati dell'AWT possono generare questa eccezione:

- ✔ `UnsupportedFlavorException` (dati in formato non appropriato).

Eccezioni di java.beans

Le classi del package `java.beans` generano le seguenti eccezioni:

- ✔ `IntrospectionException` (impossibile risolvere l'oggetto durante l'introspezione);
- ✔ `PropertyVetoException` (modifica delle proprietà non legale).

Eccezioni di java.io

Le classi nel package `java.io` generano diverse eccezioni, come indicato nella Tabella 6.1 e nella Figura 6.1. Tutte le classi che lavorano con l'I/O sono soggette a generare eccezioni correggibili. Ad esempio, attività quali l'apertura di file o la scrittura su file a volte possono non giungere a buon fine. Le classi del package `java.io` non generano errori.

Figura 6.1
La gerarchia di eccezioni di java.io.

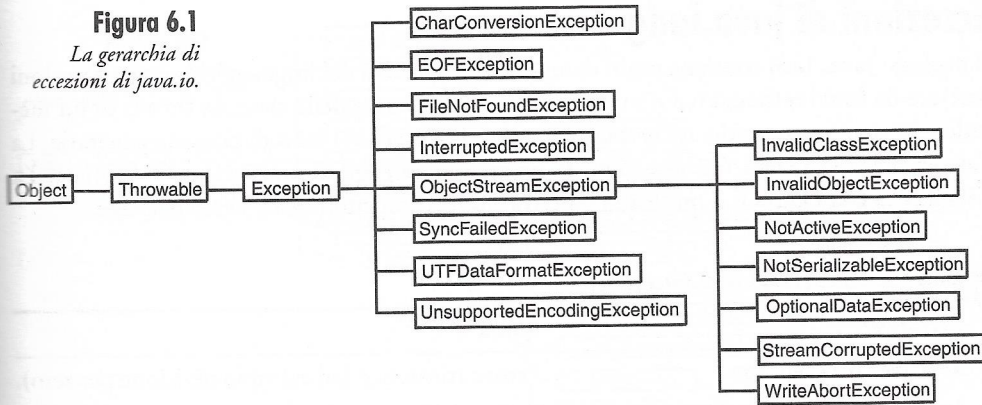



Tabella 6.1 *Le eccezioni di java.io.*

Eccezione	*Causa*
`CharConversionException`	Classe radice per le eccezioni di conversione dei caratteri.
`IOException`	Classe radice per le eccezioni di I/O.
`EOFException`	Fine del file.
`FileNotFoundException`	Impossibile individuare il file.
`InterruptedIOException`	L'operazione di I/O è stata interrotta; contiene un membro `bytesTransferred` che indica quanti byte sono stati trasferiti prima che l'operazione fosse interrotta.
`InvalidClassException`	La classe non è valida per la serializzazione.
`InvalidObjectException`	La classe impedisce esplicitamente la serializzazione.
`NotActiveException`	Serializzazione non attiva.
`NotSerializableException`	La classe non può essere serializzata.
`ObjectStreamException`	Classe radice per le eccezioni del flusso di oggetti.
`OptionalDataException`	Contiene membri di dati per indicare la fine del file o dati opzionali da leggere.
`StreamCorruptedException`	Il flusso non ha passato la verifica di coerenza interna.
`SyncFailedException`	Sincronizzazione non terminata.
`UTFDataFormatException`	Stringa UTF-8 malformata.
`UnsupportedEncodingException`	Meccanismo di codifica dei caratteri non supportato.
`WriteAbortException`	Eccezione nel flusso.

Eccezioni di java.lang

Il package `java.lang` contiene molti componenti essenziali del linguaggio Java. Le eccezioni derivate da `RuntimeException` non devono essere dichiarate nella clausola `throws` di un metodo, ma vengono considerate *normali*, in quanto quasi tutti i metodi possono generarle. La Tabella 6.2 e la Figura 6.2 mostrano le eccezioni correggibili nel package `java.lang`. La Tabella 6.3 e la Figura 6.3 mostrano gli errori non correggibili nello stesso package.

Tabella 6.2 *Le eccezioni di java.lang.*

Eccezione	*Causa*
`ArithmeticException`	Errore aritmetico (ad esempio divisione per zero).
`ArrayIndexOutOfBoundsException`	Indice dell'array inferiore a zero o superiore alle dimensioni reali dell'array.
`ArrayStoreException`	Il tipo di oggetto non corrisponde all'array e all'oggetto da memorizzare nell'array.
`ClassCastException`	Casting di oggetto in tipo non appropriato.
`ClassNotFoundException`	Impossibile caricare la classe richiesta.
`CloneNotSupportedException`	L'oggetto non implementa l'interfaccia clonabile.
`Exception`	Classe radice della gerarchia di eccezioni.
`IllegalAccessException`	La classe non è accessibile.
`IllegalArgumentException`	Il metodo ha ricevuto un argomento non lecito.
`IllegalMonitorStateException`	Stato improprio del monitor (sincronizzazione dei thread).
`IllegalStateException`	Metodo richiamato in un momento errato.
`IllegalThreadStateException`	Il thread è in uno stato non appropriato per l'operazione richiesta.
`IndexOutOfBoundsException`	L'indice è al di fuori dei limiti.
`InstantiationException`	*Tentativo di creare un'istanza da una classe astratta.*
`InterruptedException`	Thread interrotto.
`NegativeArraySizeException`	Dimensioni dell'array inferiori a zero.
`NoSuchFieldException`	Tentativo di accedere a campi non validi.
`NoSuchMethodException`	Impossibile risolvere il metodo.
`NullPointerException`	Tentativo di accedere a membri di oggetti null.
`NumberFormatException`	Impossibile convertire la stringa in numero.
`RuntimeException`	Classe di base per molte eccezioni di `java.lang`.
`SecurityException`	Operazione non consentita dale impostazioni di sicurezza.
`StringIndexOutOfBoundsException`	L'indice è negativo o superiore alle dimensioni della stringa.

Figura 6.2
La gerarchia di eccezioni di java.lang.

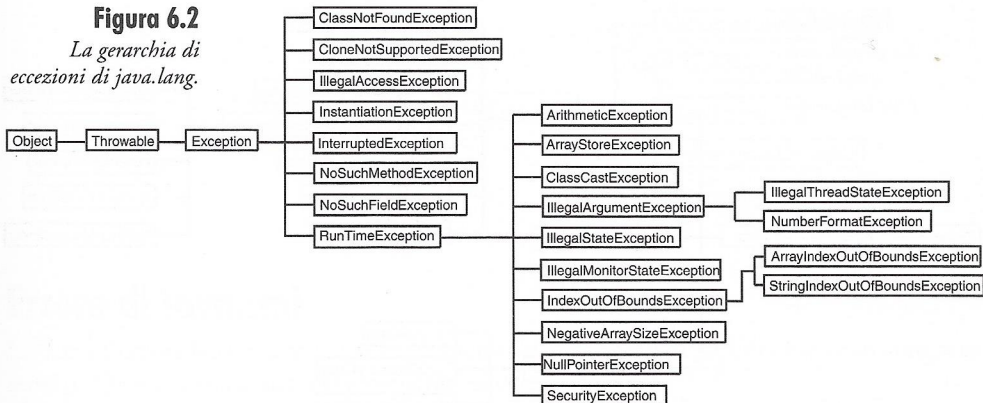
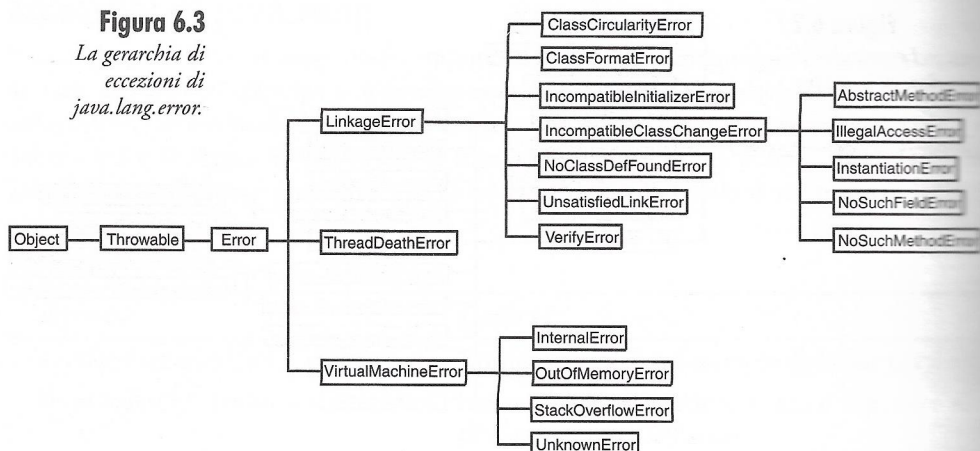



Tabella 6.3 *Gli errori di java.lang.*

Errore	*Causa*
`AbstractMethodError`	Tentativo di richiamare un metodo astratto.
`ClassCircularityError`	Questo errore non viene più utilizzato.
`ClassFormatError`	Formato di classe binario non valido.
`Error`	Classe radice della gerarchia di errori.
`ExceptionInInitializerError`	Eccezione imprevista nell'inizializzatore.
`IllegalAccessError`	Tentativo di accedere a un oggetto inaccessibile.
`IncompatibleClassChangeError`	Utilizzo improprio della classe.
`InstantiationError`	Tentativo di creare un'istanza da una classe astratta.
`InternalError`	Errore nell'interprete.
`LinkageError`	Errore nella dipendenza di classe.
`NoClassDefFoundError`	Impossibile trovare la definizione di classe.
`NoSuchFieldError`	Impossibile trovare il campo richiesto.
`NoSuchMethodError`	Impossibile trovare il metodo richiesto.
`OutOfMemoryError`	Memoria esaurita.
`StackOverflowError`	Overflow nello stack.
`ThreadDeath`	Indica che il thread terminerà; può essere intercettato per eseguire la pulizia (in questo caso deve essere rigenerato).
`UnknownError`	Errore sconosciuto della macchina virtuale.
`UnsatisfiedLinkError`	Collegamenti non risolti nella classe caricata.
`VerifyError`	Impossibile verificare il bytecode.
`VirtualMachineError`	Classe radice degli errori della macchina virtuale.

Figura 6.3 *La gerarchia di eccezioni di java.lang.error.*

Eccezione di java.lang.reflect

Le classi di `java.lang.reflect` generano la seguente eccezione:

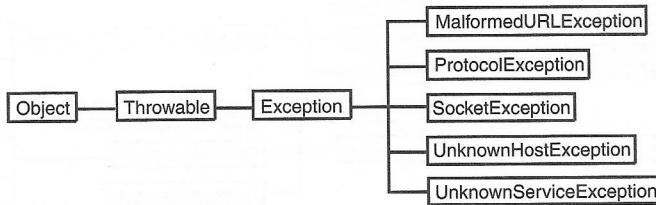
✔ `InvocationTargetException` (il metodo richiamato ha generato un'eccezione).

Eccezioni di java.net

Il package `java.net` gestisce le comunicazioni di rete. Nella maggior parte dei casi le sue classi generano eccezioni per indicare mancate connessioni e fatti simili. La Tabella 6.4 e la Figura 6.4 mostrano le eccezioni correggibili del package `java.net`. Le classi di questo package non generano errori.

Tabella 6.4 *Le eccezioni di java.net.*

Eccezione	*Causa*
`BindException`	Impossibile collegare il socket – porta in uso.
`ConnectException`	Il socket remoto ha rifiutato la connessione. Nessun socket in ricezione.
`MalformedURLException`	Impossibile interpretare l'URL.
`NoRouteToHostException`	Impossibile raggiungere l'host, firewall presente.
`ProtocolException`	Errore nel protocollo della classe del socket.
`SocketException`	Eccezione della classe del socket.
`UnknownHostException`	Impossibile risolvere il nome dell'host.
`UnknownServiceException`	La connessione non supporta il servizio.

Figura 6.4 *La gerarchia di eccezioni di java.net.*

Errore di java.rmi

Le classi Remote Method Invocation permettono agli oggetti di Java di esistere su computer remoti. Queste classi generano il seguente errore:

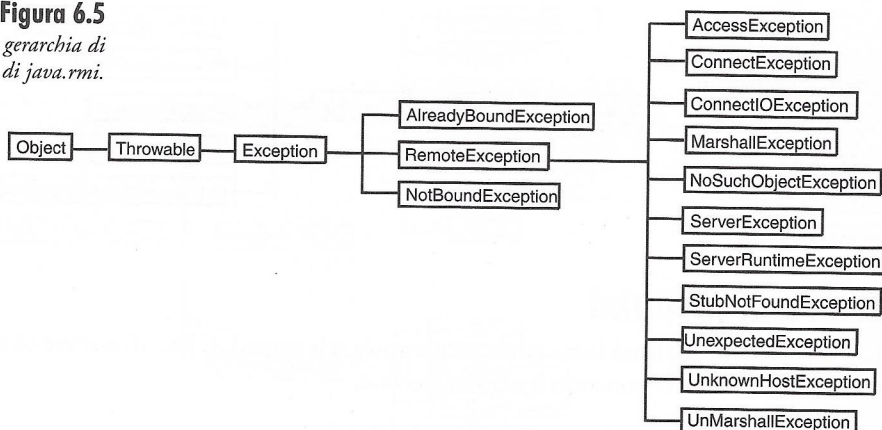
- `ServerError` (il server remoto indica un errore).

Eccezioni di java.rmi

Gli oggetti di Java i cui metodi vengono richiamati a distanza per mezzo dell'RMI possono generare eccezioni. La Tabella 6.5 e la Figura 6.5 mostrano le eccezioni generate dal package `java.rmi`.

Tabella 6.5 *Le eccezioni di java.rmi.*

Eccezione	*Causa*
`AccessException`	Operazione non permessa.
`AlreadyBoundException`	Il nome è già collegato.
`ConnectException`	L'host ha rifiutato la connessione.
`ConnectIOException`	Eccezione di I/O durante la connessione.
`MarshalException`	Errore durante il "marshaling".
`NoSuchObjectException`	Oggetto non più esistente.
`NotBoundException`	Il nome non è collegato.
`RMISecurityException`	L'`RMISecurityManager` genera un'eccezione.
`RemoteException`	Metodo remoto non valido.
`ServerException`	Il server remoto genera un'eccezione.
`ServerRuntimeException`	Il server remoto genera un'eccezione di esecuzione.
`StubNotFoundException`	Oggetto remoto non esportato.
`UnexpectedException`	Errore sconosciuto.
`UnknownHostException`	Eccezione non nella segnatura del metodo.
`UnmarshalException`	Errore in fase di "unmarshaling"; possibile corruzione del flusso.

Figura 6.5
La gerarchia di eccezioni di java.rmi.

Eccezioni di java.rmi.server

I server RMI generano eccezioni. La Tabella 6.6 mostra queste eccezioni di `java.rmi.server`.

Tabella 6.6 *Le eccezioni di java.rmi.server.*

Eccezione	*Causa*
`ExportException`	Porta in uso.
`ServerCloneException`	Impossibile clonare.
`ServerNotActiveException`	Il server non esegue il metodo remoto.
`SkeletonMismatchException`	Stub e scheletro non corrispondono.
`SkeletonNotFoundException`	Scheletro non trovato o non valido.
`SocketSecurityException`	Tentativo di utilizzare una porta non valida.

Eccezioni di java.security

L'API della sicurezza permette agli utenti di implementare in Java le funzionalità di sicurezza. L'API include il supporto per le segnature digitali, la codifica dei dati, la gestione delle chiavi e il controllo dell'accesso. La Tabella 6.7 e la Figura 6.6 mostrano le eccezioni generate dal package `java.security`.

Eccezioni di java.security.acl

L'API dell'ACL (Access Control List) di Java permette agli sviluppatori di limitare l'accesso a determinati utenti. Le classi di `java.security.acl` generano le seguenti eccezioni:

- `ACLNotFoundException` (impossibile trovare l'elenco di controllo degli accessi);
- `LastOwnerException` (tentativo di eliminare ultimo possessore dell'ACL);
- `NotOwnerException` (può essere modificato solo dal possessore).

Figura 6.6 *La gerarchia di eccezioni di java.security.*

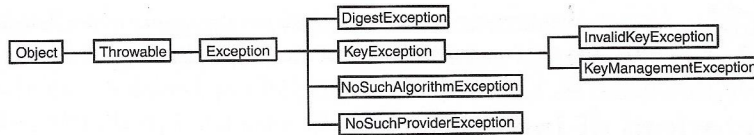



Tabella 6.7 *Le eccezioni di java.security.*

Eccezione	*Causa*
`DigestException`	Errore digest generico.
`InvalidKeyException`	Chiave non valida.
`InvalidParameterException`	Parametro del metodo non valido.
`KeyException`	Errore della chiave generico.
`KeyManagementException`	Errore del sistema di gestione delle chiavi.
`NoSuchAlgorithmException`	L'algoritmo non esiste.
`NoSuchProviderException`	Il fornitore di sicurezza non è disponibile.
`ProviderException`	Eccezione del fornitore di sicurezza.
`SignatureException`	Errore di segnatura generico.

Eccezioni di java.sql

L'API dell'SQL di Java genera le seguenti eccezioni:

- ✔ `DataTruncation` (troncamento dei dati non previsto);
- ✔ `SQLException` (errore SQL – contiene informazioni dettagliate su SQL);
- ✔ `SQLWarning` (avvertimento SQL).

Eccezione di java.text

L'API dei testi di Java genera la seguente eccezione:

- ✔ `FormatException` (errore nella formattazione o nell'analisi).

Eccezioni di java.util

Le classi del package `java.util` generano le seguenti eccezioni:

- ✔ `EmptyStackException` (nessun oggetto nello stack);
- ✔ `MissingResourceException` (risorsa non disponibile);
- ✔ `NoSuchElementException` (nessun oggetto nella collezione);
- ✔ `TooManyListenersException` (generato da ascoltatori di eventi unicast).

Unicast è un termine utilizzato in Java per un oggetto server "singleton". I singleton sono oggetti che possono essere istanziati una sola volta.

Eccezioni di java.utils.zip

L'API zip degli strumenti di utilità di Java genera le seguenti eccezioni:

- `DataFormatException` (errore del formato);
- `ZipException` (errore Zip).

Eccezioni integrate

Nell'esempio riportato nel Listato 6.4 si vede come funzionano le eccezioni *automatiche* in Java. Questa applicazione crea un metodo e lo obbliga a effettuare una divisione per zero. Non è necessario che il metodo generi esplicitamente un'eccezione, in quanto l'operatore di divisione la genera quando necessario.

Listato 6.4 *Un esempio di eccezione integrata.*

```
import java.io.* ;
import java.lang.Exception ;
public class DivideBy0 {
    public static void main( String[] args ) {
    int a = 2 ;
    int b = 3 ;
    int c = 5 ;
    int d = 0 ;
    int e = 1 ;
    int f = 3 ;
    try
        {
        System.out.println( a+"/"+b+" = "+div( a, b ) ) ;
        System.out.println( c+"/"+d+" = "+div( c, d ) ) ;
        System.out.println( e+"/"+f+" = "+div( e, f ) ) ;
        }
    catch( Exception except )
        {
        System.out.println( "Intercettata eccezione " +
                            except.getMessage() ) ;
        }
    }
    static int div( int a, int b ) {
        return (a/b) ;
    }
}
```

L'output di questa applicazione è:

```
2/3 = 0
Intercettata eccezione / by zero
```

La prima chiamata a `div()` è corretta, mentre la seconda è errata a causa dell'errore di divisione per zero. Nonostante l'applicazione non lo abbia specificato, è stata generata e intercettata un'eccezione. In questo modo è possibile utilizzare l'aritmetica nel codice senza dover scrivere codice che controlla esplicitamente i limiti.

Riepilogo

Il meccanismo di gestione delle eccezioni in Java permette ai metodi di riportare gli errori in modo che non possano essere ignorati. Ogni eccezione generata deve esser intercettata, diversamente l'applicazione termina. Le eccezioni sono oggetti di classe derivati dalla classe `Throwable` e pertanto uniscono dati e metodi; un oggetto eccezione di norma contiene una stringa di descrizione dell'errore.

La gestione delle eccezioni aiuta a raccogliere l'elaborazione degli errori in un unico luogo, unendo il riporto dei risultati e il riporto degli errori e permettendo così di creare codice molto più potente e robusto.